Évaluation rapide de fonctions hypergéométriques

Emmanuel Jeandel

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

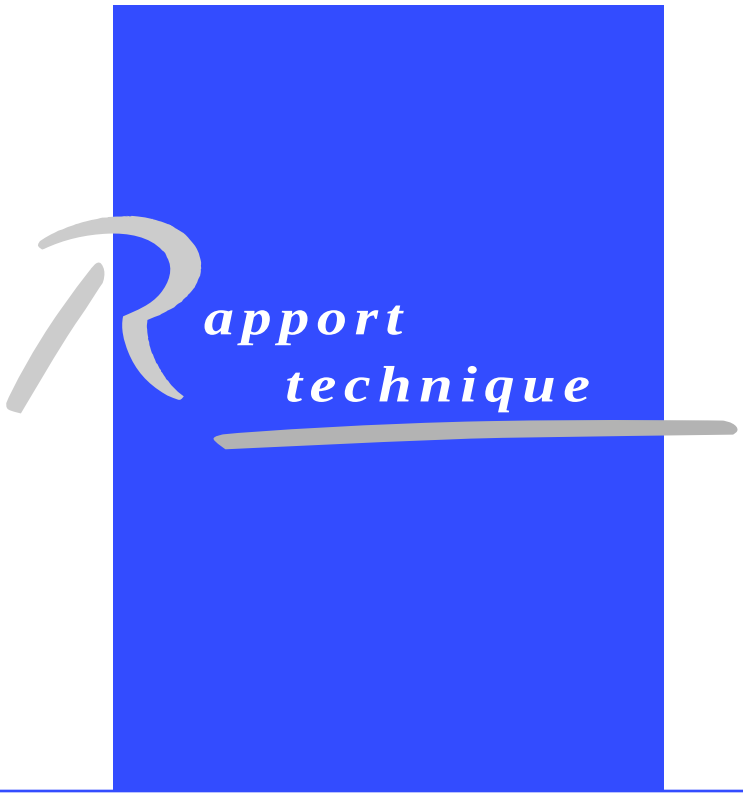
Évaluation rapide de fonctions hypergéométriques

Emmanuel Jeandel

N° 0242

Juillet 2000

THÈME 2



ISSN 0249-0803

Évaluation rapide de fonctions hypergéométriques

Emmanuel Jeandel

Thème 2 — Génie logiciel
et calcul symbolique
Projet PolKA

Rapport technique n° 0242 — Juillet 2000 — 17 pages

Résumé : Nous présentons ici l'implantation des fonctions hypergéométriques dans la bibliothèque MPFR. Ceci a été effectué à l'aide de la méthode Binary Splitting. Un algorithme générique a donc été créé, qui a permis l'amélioration de l'exponentielle, de certaines constantes, et l'implantation du sinus et du cosinus. Nous exposons l'algorithme pour le cas rationnel, puis nous montrons comment ce cas particulier permet d'obtenir l'exponentielle. Nous utilisons ensuite une méthode similaire pour les autres fonctions. Les expériences montrent que la méthode est plus efficace que celles employées précédemment dans MPFR.

Mots-clés : Fonctions hypergéométriques, Binary Splitting, bibliothèque MPFR

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Fast Evaluation of Hypergeometric Functions

Abstract: An algorithm for fast calculation of hypergeometric functions, based on the Binary Splitting method is proposed. A fast generic code is implemented, allowing to use in MPFR some transcendental functions (exp, sin, cos ...) We first consider the case of rational numbers, then we apply the results to compute the exponential of a floating point number. We use the same algorithm to compute other hypergeometric functions. This method gives good results, with a better computational complexity than previous implementations in MPFR, as shown by computers experiments.

Key-words: Hypergeometric functions, Binary Splitting, MPFR library

1 Introduction

Les imprécisions de calcul sont souvent la cause de nombreuses erreurs, et il convient de savoir les éviter. La bibliothèque mpfr [Pol99] s'est justement donné ce but, s'appuyant sur la gestion des entiers dans GMP, décrite dans [Gra00]. L'objectif de ce travail est justement d'améliorer cette bibliothèque, en lui ajoutant quelques fonctions, à l'aide des fonctions hypergéometriques, et en implantant des algorithmes basés sur la méthode "Binary Splitting". Les travaux suivants ont été réalisés :

- Amélioration du calcul de l'exponentielle, $\ln 2$ et π.
- Ajout de sin, cos.
- Création d'un code C générique pour le calcul d'une quelconque fonction hypergéométrique ${}_2\mathrm{F}_1$ avec un argument rationnel.
- Création de scripts autoconf et automake.

Nous commencerons par définir les fonctions hypergéometriques, puis nous étudierons leur implantation avec des arguments rationnels. Nous utiliserons alors cet algorithme pour calculer l'exponentielle d'un nombre réel, puis nous appliquerons une technique analogue pour calculer sin, cos, π, $\ln 2$.

2 L'algorithme

2.1 Définitions

Une fonction hypergéométrique possède un développement en série de Taylor à l'origine de la forme :

$$f(x) = \sum_0^\infty a_n x^n$$

où les coefficients vérifient :

$$\frac{a_{n+1}}{a_n} = \frac{P(n)}{Q(n)}$$

avec P et Q deux polynômes.
Ceci est un cas assez général, qui regroupe la plupart des fonctions connues. On se limitera dans la suite au cas particulier :

$$f(x) = \sum_0^\infty \frac{a(a+1)\cdots(a+n-1)b(b+1)\cdots(b+n-1)}{n!c(c+1)\cdots(c+n-1)} x^n.$$

Ce cas correspond aux fonctions hypergéométriques du type ${}_2F_1(a,b,c;z)$ (les indices indiquant les degrés des polynômes), qui s'obtiennent en prenant $P(n) = (a+n)(b+n)$ et $Q(n) = (c+n)(1+n)$ [1].

On étudiera également dans la suite des cas encore plus restreints où on a supprimé les variables $a,b\ldots$. (Il s'agit donc des fonctions ${}_1F_1$, ${}_0F_1$ et ${}_0F_0$). En effet, la plupart des fonctions courantes ($\exp, \sin, \cos$) sont obtenues à partir de ces fonctions, comme le montre le tableau 1 .

1. Le terme $(1+n)$, correspondant à la factorielle, est supposé toujours existant, et n'est donc pas considéré dans la définition des familles ${}_pF_q$.

f(x)	Représentation
$\log(x)$	$(x-1)\cdot {}_2\mathrm{F}_1\left(1,1,2;1-x\right)$
$\sinh(x)$	$x\cdot {}_0\mathrm{F}_1\left(\frac{3}{2};\frac{x^2}{4}\right)$
$\cos(x)$	${}_0\mathrm{F}_1\left(\frac{1}{2};\frac{-x^2}{4}\right)$
$\exp(x)$	${}_0\mathrm{F}_0\left(x\right)$
$\arctan(x)$	$x\cdot {}_2\mathrm{F}_1\left(1,\frac{1}{2},\frac{3}{2};-x^2\right)$

TAB. 1 – *Expression de fonctions usuelles à l'aide des* ${}_pF_q$

2.2 L'algorithme pour les rationnels

2.2.1 La méthode Binary Splitting

Commençons par décrire la méthode utilisée dans un cas simple, le calcul de la factorielle. Nous utiliserons ensuite le même procédé pour le cas qui nous intéresse. On pose

$$F(i,j) = \frac{j!}{i!} = (i+1)(i+2)\dots j.$$

Alors on s'aperçoit que

$$F(i,j) = F\left(i, \left\lfloor \frac{i+j}{2} \right\rfloor\right) F\left(\left\lfloor \frac{i+j}{2} \right\rfloor, j\right)$$

et

$$F(i,i+1) = i+1$$
$$F(i,i) = 1.$$

Et pour calculer $n! = F(0,n)$, il suffit donc d'utiliser les formules ci-dessus récursivement.

Il faut comprendre l'intérêt de la méthode de calcul : l'idée est en fait de multiplier des nombres de même taille. La méthode classique, elle, revient à multiplier un grand nombre ($(n-1)!$) par un nombre beaucoup plus petit (n); elle est donc beaucoup moins efficace : elle effectue le même nombre de multiplications, mais sur des nombres plus déséquilibrés. La méthode binary splitting sera donc plus efficace, dès que la multiplication utilise un algorithme non quadratique (Karatsuba ou FFT, par exemple).

Calculons en effet la complexité. On effectue une multiplication de deux nombres de taille $m/2$, puis 2 multiplications de nombres de taille $m/4$, etc ..., où m désigne la taille de $n!$ ($m \equiv n\log n$) . Ainsi, la complexité en temps est de :

$$\sum_{i=0} 2^k M\left(\frac{m}{2^{k+1}}\right)$$

où $M(x)$ désigne le coût d'une multiplication de 2 nombres de taille x.

Ainsi, avec une méthode à la Karatsuba ($M(n/2) = \alpha M(n)$), la complexité est alors de :

$$\frac{\alpha}{1-2\alpha} M(m).$$

Avec la FFT, ($M(n) = n\log n$), la complexité est alors en

$$M(n) = O(M(m)\log m).$$

2.2.2 L'algorithme

L'idée est de calculer tous les coefficients dans un certain ordre (voir [Kar97]), en mettant les dénominateurs en commun. On pose pour cela :

$$S(i,j) = \sum_{n=2^i j+1}^{2^i(j+1)} \frac{(a)\cdots(a+n-1)(b)\cdots(b+n-1)}{n!(c)\cdots(c+n-1)} x^n$$

$$\zeta(i,j) = \prod_{n=2^i j+1}^{2^i(j+1)} n(c+n-1)$$

$$\delta(i,j) = \prod_{n=2^i(j-1)+1}^{2^i j} (a+n-1)(b+n-1)$$

$$\begin{aligned}\alpha(i,j) &= S(i,j)\frac{(2^i(j+1))!\prod_{n=1}^{2^i(j+1)}(c+n-1)}{x^{2^i(j+1)}\prod_{n=1}^{2^i j}(a+n-1)(b+n-1)} \\ &= S(i,j)\frac{\prod_{n=0}^{j}\zeta(i,n)}{x^{2^i(j+1)}\prod_{n=1}^{j}\delta(i,n)}\end{aligned}$$

On a alors les résultats suivants :

$$\begin{aligned}\alpha(0,j) &= (a+j)\,(b+j) \\ \zeta(i+1,j) &= \zeta(i,2j)\,\zeta(i,2j+1) \\ \delta(i+1,j) &= \delta(i,2j)\,\delta(i,2j-1) \\ S(i+1,j) &= S(i,2j) + S(i,2j+1) \\ \alpha(i+1,j) &= \alpha(i,2j+1)\,\delta(i,2j+1) + x^{2^i}\,\zeta(i,2j+1)\,\alpha(i,2j)\end{aligned}$$

Ceci nous donne donc une méthode récursive de calcul, son inconvénient majeur étant l'encombrement mémoire. Il ne reste ensuite, si on veut calculer les 2^m premiers coefficients de la somme, qu'à calculer

$$S(m,0) = \alpha(m,0)x^{2^m}/\zeta(m,0)$$

On remarquera dans cette méthode que l'on a la propriété suivante : δ, ζ, et α sont entiers, si a,b,c et x le sont.

Cet algorithme s'ajuste facilement au cas rationnel, en posant :

$$a = \frac{a_1}{a_2}, b = \frac{b_1}{b_2}, c = \frac{c_1}{c_2}, x = \frac{p}{q}$$

2.2.3 Code en Maple

L'algorithme Maple se déduit des considérations précédentes.

```
Hypergeom21 := proc(a1, a2, b1, b2, c1, c2, p, q, m)
  local n, ζ, δ, α, j, l, qa2b2toj, pc2toj, i;
    n := 2^m;
    pc2toj := p × c2;
    qa2b2toj := q × a2 × b2;
    ζ := [seq(i × (c1 + c2 × (i − 1)), i = 1..n)];
    δ := [seq((a1 + a2 × i) × (b1 + b2 × i), i = 0..n − 1)];
    α := map(`*`, [seq((1 + i) × (c1 + c2 × i), i = 0..n − 1)], qa2b2toj);
    j := 1;
    l := n;
    while j < n do
      l := 1/2 × l;
      α := [seq(qa2b2toj × ζ_{2i} × α_{2i−1} + pc2toj × δ_{2i−1} × α_{2i}, i = 1..l)];
      j := 2 × j;
      qa2b2toj := qa2b2toj^2;
      ζ := [seq(ζ_{2i−1} × ζ_{2i}, i = 1..l)];
      if j < n then pc2toj := pc2toj^2; δ := [seq(δ_{2i−1} × δ_{2i}, i = 1..l)] fi
    od;
    evalf(α_1/(P_1 × qa2b2toj))
  end
```

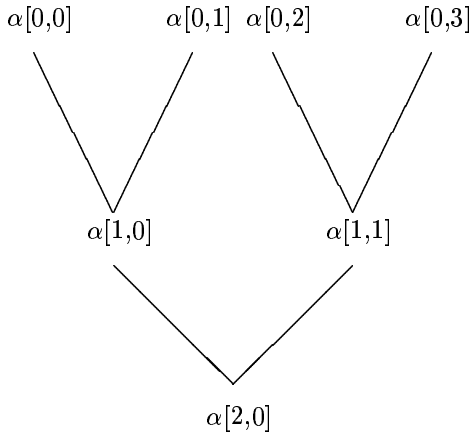
3 Application à l'exponentielle ; le cas réel

Nous allons détailler le cas de l'exponentielle, qui est plus simple que le cas général, puisqu'on peut éliminer les variables a, b et c.

3.1 Dérécursivation

On va maintenant essayer de dérécursiver l'algorithme. Cette méthode est également applicable dans le cas général, mais on ne détaillera les calculs que dans ce cas.

Les calculs se font normalement suivant un arbre :



On va alors changer l'ordre d'évaluation des termes, en suivant l'idée suivante, valable pour l'arbre précédent :

- On commence par calculer $\alpha(0,0)$.
- Une fois qu'on a calculé $\alpha(0,1)$, on peut calculer $\alpha(1,0)$.
- On calcule $\alpha(0,2)$.
- On calcule $\alpha(0,3)$, qui nous permet de calculer $\alpha(1,1)$, puis $\alpha(2,0)$.

L'ordre des calculs est différent, mais les calculs effectués sont similaires, donc la complexité combinatoire sera la même.

Plus formellement, on pose, pour $n \geq 1$

$$\beta(n,k) = \prod_{i=\lfloor \frac{n-1}{2^k} \rfloor 2^k+1}^{\lfloor \frac{n}{2^k-1} \rfloor 2^{k-1}} (1+i)$$

et

$$\alpha(n,k) = \sum_{i=\lfloor \frac{n-1}{2^k} \rfloor 2^k+1}^{\lfloor \frac{n}{2^k-1} \rfloor 2^{k-1}} n! \frac{x^i}{i!}$$

On suppose calculé $\alpha(n,k)$ et $\beta(n,k)$, $k = 1 \ldots$. Soit m la plus grande puissance de 2 qui divise $n+1$, c'est-à-dire : $n+1 = 2^m(2j+1)$. On a alors :

$$\forall k \in [1..m] \begin{cases} \alpha(n+1,k) = \alpha(n+1,k-1) + \beta(n+1,k-1)\alpha(n,k) \\ \beta(n+1,k) = \beta(n,k) * \beta(n+1,k-1) \end{cases}$$

$$\begin{cases} \alpha(n+1,m+1) = \alpha(n,m) \\ \beta(n+1,m+1) = \beta(n,m) \end{cases}$$

Avec pour convention

$$\begin{cases} \alpha(n+1,0) = p(n+1) \\ \beta(n+1,0) = q(n+1) \end{cases}$$

L'algorithme Maple s'appuyant sur la remarque est alors le suivant :

```
Essai := proc(p, q, m)
    local i, α, β, j, acc_β, acc_α, tmp, k;
      i := 2^m ;
      α := array(0..m) ;
      β := array(0..m) ;
      for j to i do
          acc_β := q × j ;
          acc_α := p^j ;
          tmp := j ;
          k := 0 ;
          while tmp mod 2 = 0 do
              β_k := β_k × acc_β ;
              α_k := acc_α + acc_β × α_k ;
              acc_β := β_k ;
              acc_α := α_k ;
              tmp := 1/2 × tmp ;
              k := k + 1
          od;
          β_k := acc_β ;
          α_k := acc_α
      od;
      α_m/β_m
    end
```

3.2 Le cas réel

Les algorithmes précédents sont valables pour des rationnels, et, ils ne sont pas efficaces pour des réels. En effet, dans ce cas, toutes les quantités avec lesquelles on travaille sont des flottants sur n bits, et le moindre calcul se fera donc avec des nombres de n bits. On perd donc tout l'intérêt de la méthode. Ainsi, pour calculer l'exponentielle, on a recours à une astuce de calcul due à Brent [Bre76] (Voir aussi [HP97]). L'idée est simple. On écrit x sous la forme :

$$x = x_0 + \sum_{k=0}^{\infty} \frac{u_k}{v_k}$$

avec x_0 entier, $v_k = 2^{2^k}$, $u_k < 2^{2^{k-1}}$. On se sert ensuite de la formule d'addition de l'exponentielle pour obtenir le bon résultat.
Montrons le découpage sur un exemple :

$$x = \frac{199}{256} = 10111101 \cdot 2^{-8}$$

$$x = \boxed{1}\,\boxed{0}\,\boxed{1\ 1}\,\boxed{1\ 1\ 0\ 1}$$

$$x = \frac{1}{2^1} + \frac{0}{2^2} + \frac{3}{2^4} + \frac{11}{2^8}$$

Essayons de comprendre pouquoi cette version est meilleure ; si on effectue l'algorithme avec des réels, il est nécessaire de faire tous les calculs avec une précision maximum. Mais dans cette démarche, l'astuce est qu'il suffit pour les nombres faibles en valeur absolue (k important) de faire moins de calcul; En particulier, le nombre de termes qui importent est beaucoup plus faible. Ainsi, si on veut calculer le résultat avec une précision 2^n (i.e. n bits de mantisse), on aura besoin, pour obtenir $\exp(u_k/v_k)$, de lancer l'algorithme avec $m = n - k$. Ainsi, on se rend compte que les deux contributions se compensent, et que l'algorithme sera ainsi meilleur. Chaque étape se réalise donc en $O(M(n)\log n)$, et l'algorithme sera ici en $O(M(n)\log^2 n)$, avec l'algorithme FFT pour la multiplication.

3.3 Analyse d'erreur

3.3.1 Erreur produite par une multiplication

On va calculer l'erreur produite par la multiplication de deux nombres a et b proches de 1 (on peut toujours s'y ramener, sans rien changer aux calculs d'erreur), et obtenus avec des erreurs $\mathrm{err}(a)$ et $\mathrm{err}(b)$. On dispose donc de a' et b', tels que

$$\begin{cases} |a' - a| < \mathrm{err}(a) \\ |b' - b| < \mathrm{err}(b) \\ |b'| < 1 \\ |a| < 1 \end{cases} \tag{1}$$

Alors

$$\begin{array}{lcl} a'b' - ab & = & (a' - a)b' + (b' - b)a \\ |a'b' - ab| & \leq & |(a' - a)b'| + |(b' - b)a| \\ |a'b' - ab| & \leq & |a' - a| + |b' - b| \end{array} \tag{2}$$

L'erreur sur le calcul de $a'b'$ étant de $1\,\mathrm{ulp}$ [2] on a donc le résultat suivant et ses corollaires :

Erreur sur la multiplication

> Si a et b sont obtenus avec des erreurs respectives d'au plus $\mathrm{err}(a)$ et $\mathrm{err}(b)$, alors ab est obtenu avec une erreur d'au plus $\mathrm{err}(b) + \mathrm{err}(a) + 1\,\mathrm{ulp}$

2. **ulp** : unit in the last place, désigne le dernier bit représentatif dans un flottant. On note d'habitude $\mathrm{ulp}(a)$, puisque celui-ci dépend du flottant en question. Dans la suite, la plupart des flottants étant inférieurs à 1, on notera juste ulp pour $\mathrm{ulp}(1)$.

Cumul d'erreurs

- Si $a_1, \dots a_n$ sont obtenus avec une erreur de $\mathrm{err}(a_i)$ alors $a_1 \dots a_n$ est obtenu avec une erreur de $\sum \mathrm{err}(a_i) + (n-1)\,\mathrm{ulp}$
- Si a est obtenu avec une erreur de $\mathrm{err}(a)$ alors,
  - a^2 est obtenu avec une erreur de $2\,\mathrm{err}(a) + 1\,\mathrm{ulp}$
  - a^{2^n} est obtenu avec une erreur de $2^n\,\mathrm{err}(a) + (2^n - 1)\,\mathrm{ulp}$

3.3.2 La partie rationnelle

Il s'agit ici du calcul le plus simple : On s'intéresse à l'erreur commise en appelant
`mpfr_exp_rational(y,p,r,m)`
qui calcule les 2^m premiers termes de l'exponentielle de p/q, ($q = 2^{2^k}$, $r = 2^k$), avec une précision correspondante à celle de y. Pour vérifier que ce calcul nous donne bien l'exponentielle recherchée, il suffit de vérifier que l'on fait suffisament de calculs ... .

Lorsqu'on appelle cette fonction, on sait que p est majoré par $2^{2^{k-1}}$. Le terme négligé est $\sum_{i \geq 2^m} \frac{p^i}{q^i i!}$, que l'on peut majorer par $2\frac{p^{2^m}}{q^{2^m}(2^m)!}$. On veut donc

$$2\left(\frac{p}{q}\right)^{2^m} \frac{1}{2^m!} < \frac{1}{2^{prec}}$$

Il suffit d'avoir

$$\left(\frac{1}{2^{2^{k-1}}}\right)^{2^m} < \frac{1}{2^{prec}}$$

Ou encore

$$2^{2^{k-1}2^m} > 2^{prec}$$

Il suffit donc de prendre

$$m > \log_2(prec) - k + 1.$$

C'est bien ce qui est transmis en pratique, et donc le bon calcul est mené : il est certain que les termes négligés sont bien négligeables. Comme tout le calcul se fait avec des entiers, l'erreur commise sur le résultat est minime : elle se résume à 2 ulp : un pour la troncature, et un pour la division.

Calcul de l'exponentielle

L'erreur commise par le calcul de l'exponentielle d'un nombre rationnel est de 2 ulp.

3.3.3 La boucle principale

Maintenant que l'on connaît les erreurs dans l'évaluation de l'exponentielle rationnelle, on peut détailler le cas réel; Les erreurs proviennent de trois sources :

- Pour les réels supérieurs à 1, on opère une réduction d'argument, ce qui nécessite des élévations au carré en fin d'algorithme. Ainsi, si on appelle $\mathrm{shift}(x) = \max(\lfloor \log_2(x) \rfloor, 0)$, il faut effectuer $\mathrm{shift}(x)$ élévations au carré.
- Pour le premier terme, celui-ci est également décalé de `BMPL`/2, où `BMPL` désigne la taille du mot machine (32 ou 64), et on a autant d'élévations au carré.
- Les multiplications entre tous les facteurs calculés : il y en a exactement $\lceil \log_2(\mathrm{prec}(x)) \rceil - \log_2(\texttt{BMPL})$.[3]

Pour trouver l'erreur commise, il faut prendre toutes ces opérations dans l'ordre dans lequel elles s'exécutent

3. Le deuxième facteur vient du fait que l'algorithme utilise un découpage en mots machines et non en bits.

1. L'algorithme commence par le calcul du premier terme $\exp(u_0/v_0)$. Celui-ci se calcule en deux temps : On commence par calculer l'exponentielle du rationnel, ce qui occasionne une erreur de 2 ulp, puis on effectue $\texttt{BMPL}/2$ élévations au carré. L'erreur commise est donc de :

$$2^{\texttt{BMPL}/2} - 1 + 2^{\texttt{BMPL}/2}\,\mathrm{err}(\exp(u_0))$$

Ou encore

$$3 \cdot 2^{\texttt{BMPL}/2} - 1$$

2. Il y a ensuite les $\log_2(\mathrm{prec}(x)) - \log_2(\texttt{BMPL})$ multiplications. Là aussi, il faut d'abord calculer toutes les exponentielles, ce qui provoque une erreur de 2 ulp pour chaque facteur, excepté le premier, puis calculer les produits. L'erreur totale est donc de :

$$(\log_2(\mathrm{prec}(x)) - \log_2(\texttt{BMPL})) + (\log_2(\mathrm{prec}(x)) - \log_2(\texttt{BMPL})) * 2 + 3 \cdot 2^{\texttt{BMPL}/2} - 1$$

ou encore

$$3(\log_2(\mathrm{prec}(x)) - \log_2(\texttt{BMPL}) + 2^{\texttt{BMPL}/2}) - 1$$

3. Il reste à obtenir le résultat final. Il faut faire pour cela $\mathrm{shift}(x)$ élévations au carré, à partir du résultat obtenu dans l'étape précédente. L'erreur totale est donc de :

$$(3(\log_2(\mathrm{prec}(x)) - \log_2(\texttt{BMPL}) + 2^{\texttt{BMPL}/2}) - 1)2^{\mathrm{shift}(x)} + 2^{\mathrm{shift}(x)} - 1$$

ou encore

$$(3(\log_2(\mathrm{prec}(x)) - \log_2(\texttt{BMPL}) + 2^{\texttt{BMPL}/2}))2^{\mathrm{shift}(x)} - 1$$

Qui peut se majorer par

$$2^{\texttt{BMPL}/2+2+\mathrm{shift}(x)}\,\mathrm{ulp}$$

sous la condition :

$$3(\log_2(\mathrm{prec}(x)) - \log_2(\texttt{BMPL}) < 2^{\texttt{BMPL}/2}$$

qui est vérifiée en pratique...

4 Les autres fonctions et constantes

4.1 Implantation de l'algorithme

Pour gérer toutes les fonctions, nous avons décidé de créer une fonction générique, que l'on inclut ensuite dans le programme qui doit calculer les termes. Les diverses options suivantes ont été gérées :

- La définition des paramètres a,b,c.
- La factorielle au numérateur peut être changée.
- Le dénominateur peut être une puissance de 2 ou un entier quelconque.

Le tableau 2 illustre l'usage des paramètres dans les exemples étudiés.

$f(x)$	$P(n)/Q(n)$	Code C
$f(x) = \exp(x)$	$\frac{1}{n+1}$	`#define GENERIC mpfr_exp_aux` `#include "generic.c"`

$f(-x^2) = \cos(2x)$	$\frac{1}{(n+1)(n+\frac{1}{2})}$	`#define C`
		`#define C1 1`
		`#define C2 2`
		`#define GENERIC mpfr_cos_aux`
		`#include "generic.c"`

$f(-x^2) = \frac{\sin(2x)}{2x}$	$\frac{1}{(n+1)(n+\frac{3}{2})}$	`#define C`
		`#define C1 3`
		`#define C2 2`
		`#define GENERIC mpfr_sin_aux`
		`#include "generic.c"`

$f(x) = -\frac{\ln(1-x)}{x}$	$\frac{n+1}{n+2}$	`#define A`
		`#define A1 1`
		`#define A2 1`
		`#define C`
		`#define C1 2`
		`#define C2 1`
		`#define NO_FACTORIAL`
		`#define GENERIC mpfr_aux_log2`
		`#include "../generic.c"`

$f(-x^2) = \frac{\arctan(x)}{x}, x \in \mathbf{Q}$	$\frac{n+\frac{1}{2}}{n+\frac{3}{2}}$	`#define A`
		`#define A1 1`
		`#define A2 2`
		`#define C`
		`#define C1 3`
		`#define C2 2`
		`#define GENERIC mpfr_aux_pi`
		`#define R_IS_RATIONAL`
		`#define NO_FACTORIAL`
		`#include "../generic.c"`

TAB. 2 – *Formules génériques pour les fonctions usuelles*

4.2 Sinus et cosinus

Le procédé de calcul est ici très similaire au cas de l'exponentielle : on calcule $\cos(a_i/b_i)$ et $\sin(a_i/b_i)$ pour des rationnels, puis on applique la formule

$$\cos(a+b) = \cos a \cos b - \sin a \sin b$$

pour obtenir le cas réel. La seule difficulté est ici le calcul d'erreur.

4.2.1 Formules générales

On démontre ici des résultats qui vont servir à calculer finement les erreurs.

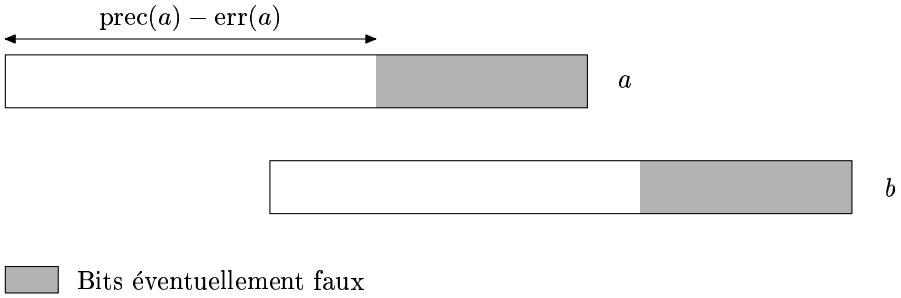
Addition, Soustraction On part de deux nombres a et b, a proche de 1 et b proche de 0, (c'est-à-dire d'exposant -1 et $-n$, avec $n > 2$ suffisament grand),obtenus avec des erreurs $\mathrm{err}(a)$ et $\mathrm{err}(b)$, et dont le nombre de chiffres est $\mathrm{prec}(a) = \mathrm{prec}(b)$. Alors, sous la condition

$$-\log_2 b + \mathrm{prec}(b) - \mathrm{err}(b) > \mathrm{prec}(a)$$

ou encore, dans le cas qui nous intéresse ($\mathrm{prec}(a) = \mathrm{prec}(b)$) :

$$\mathrm{err}(b) < -\log_2 b$$

l'erreur commise est au plus de $\mathrm{err}(a) + 2\,\mathrm{ulp}$. En effet, on se trouve alors dans la situation suivante :



Dans ce cas, l'erreur vient uniquement de a, les deux ulps venant de l'erreur sur b (au maximum 1 ulp) et de l'erreur sur l'addition.

Formule d'addition On part de $\cos a, \sin a, \cos b, \sin b$. Les erreurs se cumulent de la façon suivante :

Calcul de $\cos(a+b)$ Il faut calculer $\cos a \cos b$ et $\sin a \sin b$, puis ensuite faire la différence. L'erreur obtenue ici est donc de $3\,\mathrm{ulp} + \mathrm{err}(\cos a) + \mathrm{err}(\cos b)$, si on peut négliger la participation de $\sin a \sin b$.

Calcul de $\sin(a+b)$ Il faut calculer $\sin a \cos b$ et $\cos a \sin b$, puis faire la somme. L'erreur obtenue ici est donc de $3\,\mathrm{ulp} + \mathrm{err}(\sin a) + \mathrm{err}(\cos b)$, en négligeant les erreurs du second terme (car $|\cos a \sin b| < \sin(\mathrm{a} + \mathrm{b})$).

On en déduit le cas un peu plus général

cos(a + b)

Si on dispose de $\cos(a_i)$ et $\sin(a_i)$, avec des erreurs de $\mathrm{err}\cos(a_i)$ et $\mathrm{err}\sin(a_i)$,
- l'erreur sur $\cos(\sum_{i=1}^{i=n} a_i)$ est de $3(n-1) + \sum_{i=1}^{i=n} \mathrm{err}\cos(a_i)$
- l'erreur sur $\sin(\sum_{i=1}^{i=n} a_i)$ est de $3(n-1) + \sum_{i=2}^{i=n} \mathrm{err}\cos(a_i) + \mathrm{err}\sin(a_1)$

Formule du double Tout se passe bien dans un premier temps, puisqu'on se place dans un domaine où $|\cos x| < 1/2$, et on est sûr que l'on ne perd pas de précision à cause de la soustraction. En effet, on court souvent le risque d'avoir à effectuer une opération du type $a - b$, avec a très proche de b, ce qui occasionne une perte de précision. Par exemple, si on travaille avec 52 chiffres, avec $a = 1 + 1e - 50$ et $b = 1 + 9e - 51$, $a - b = 1e - 51$, et la précision sur le résultat n'est plus que de 1 chiffre.

Cosinus On utilise la formule $\cos 2a = 2\cos^2 a - 1$. Le calcul du carré occasionne une erreur de $2\,\mathrm{err}(\cos a) + 1\,\mathrm{ulp}$, le décalage n'occasionne pas d'erreur, et la soustraction occasionne uniquement une erreur de 1 ulp, puisqu'on est dans le cas où le deuxième opérande est connu sans erreur, et où les deux opérandes sont suffisamment différents (les exposants ont plus d'une unité de différence) pour qu'il n'y ait pas de perte de précision dans la soustraction.

Sinus Ici, le calcul $\sin 2a = 2\sin a \cos a$ est plus simple : il y a juste une multiplication, et un décalage, ce qui fait une erreur de $1\,\mathrm{ulp} + \mathrm{err}(\sin a) + \mathrm{err}(\cos a)$. On en déduit le corollaire :

Erreur dans cosinus et sinus

Si on connaît $\sin a$ et $\cos a$ avec leurs erreurs, alors :
- L'erreur dans le calcul de $\sin 2^n a$ est de $(2^n - 1)(1 + \mathrm{err}\cos a) + \mathrm{err}\sin a$
- L'erreur dans le calcul de $\cos 2^n a$ est de $2^n(1 + \mathrm{err}\cos a) - 1$.

Problème de soustraction On arrive ici au cas le plus délicat : si a est proche de $\pi/4$, le calcul de $2\cos^2 a - 1$ résultera en une perte de précision. Dans le programme, ce cas sera géré à part : Si on s'aperçoit que le calcul conduit à un problème de ce type, il suffit d'augmenter la précision du nombre de digits supprimés par l'opération, et de refaire les calculs. Ainsi, on ne tiendra pas compte de cette possibilité dans les calculs.

4.2.2 Calcul d'erreur

Reprenons un plan analogue à celui utilisé pour l'exponentielle.

1. Calculons tout d'abord l'erreur provenant du calcul du cosinus et sinus rationnels. Pour le cosinus, celle-ci se résume à 2 ulp. Pour le sinus, il faut multiplier le résultat des opérations par un facteur x, puisque les calculs sont menés en fait avec $1 - \frac{x^2}{3!} + \frac{x^4}{5!}$, et non avec $x - \frac{x^3}{3!} + \frac{x^5}{5!}$. L'erreur se résume donc ici à 3 ulp. Il serait possible de diminuer l'erreur à 2 ulp, en effectuant la multiplication par x dans la partie rationnelle de l'algorithme, et non après.
2. L'algorithme commence par le calcul du premier terme. Celui-ci se calcule en deux temps : On commence par calculer les cosinus et sinus du rationnel, ce qui occasionne une erreur de 3 ulp, puis on effectue `BMPL`/2 doubles. L'erreur commise est donc de :
$$\begin{cases} \mathrm{err}(\cos) = 3 \cdot 2^{\mathtt{BMPL}/2} - 1 \\ \mathrm{err}(\sin) = 3 \cdot 2^{\mathtt{BMPL}/2} \end{cases}$$
3. Il y a ensuite les $\log_2(\mathrm{prec}(x)) - \log_2(\mathtt{BMPL})$ formules d'addition. L'erreur totale est donc de :
$$\begin{cases} \mathrm{err}(\cos) = 3 \cdot 2^{\mathtt{BMPL}/2} + 5(\log_2(\mathrm{prec}(x)) - \log_2(\mathtt{BMPL}) - 1) + 1 \\ \mathrm{err}(\sin) = 3 \cdot 2^{\mathtt{BMPL}/2} + 5(\log_2(\mathrm{prec}(x)) - \log_2(\mathtt{BMPL}) - 1) + 1 \end{cases}$$

On s'apercoit alors que le même majorant que précédemment est valable, si le nombre dont on veut le cosinus est inférieur à un. Sinon, il peut être nécessaire de rajouter des digits.

4.3 $\ln 2$

Les choses sont ici beaucoup plus simples, puisqu'il y a moins d'opérations. En fait, la seule difficulté est de trouver une formule adéquate pour le calcul, c'est-à-dire une formule suffisamment rapide. Pour cela, on utilise l'algorithme pour calculer $\ln(1+x)$, et on essaie d'avoir x relativement petit. Les différentes formules trouvées par l'auteur sont les suivantes :

$$\begin{aligned}
\ln 2 &= -\ln\left(1-\frac{1}{2}\right)\\
\ln 2 &= -\ln\left(1-\frac{1}{4}\right)-\ln\left(1-\frac{1}{3}\right)\\
\ln 2 &= \ln\left(1+\frac{1}{8}\right)-2\ln\left(1-\frac{1}{4}\right)\\
-\ln 2 &= 15\ln\left(1-\frac{1}{16}\right)-5\ln\left(1-\frac{3}{128}\right)-3\ln\left(1-\frac{13}{256}\right)
\end{aligned}$$

La dernière formule est la plus efficace, puisque le plus grand nombre à considérer est $1/16$, ce qui assure une convergence rapide. En fait, ce qui joue le plus dans la vitesse de calcul, c'est le moment à partir duquel on peut commencer à négliger les termes. Ainsi, pour le logarithme, l'équation à vérifier est :

$$\frac{x^{2^m}}{2^m} < \frac{1}{2^{prec}}$$

La condition se traduit alors en

$$m > \log_2(prec) - \log_2(\log_2(1/x))$$

On comprend ainsi pourquoi les formules ci-dessus s'améliorent, puisque le facteur $\log_2(\log_2(1/x))$ vaut respectivement 0, 1, 1 et 2. On voit également pourquoi il est difficile de trouver une formule qui soit encore meilleure que les précédentes, puisqu'il faudrait que le plus grand facteur soit de l'ordre de $\frac{1}{256}$, ce qui rend la recherche de telles combinaisons assez pénible.

4.4 π

Le raisonnement est le même que précedemment, et il faut trouver une façon rapide de calculer. Les solutions suivantes sont envisageables :

$$\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right)-\arctan\left(\frac{1}{239}\right) \qquad \text{Machin, 1706}$$

$$\frac{\pi}{4} = 12\arctan\left(\frac{1}{18}\right)+8\arctan\left(\frac{1}{57}\right)-5\arctan\left(\frac{1}{239}\right) \qquad \text{Gauss, 1863}$$

$$\begin{aligned}
\frac{\pi}{4} &= 61\arctan\left(\frac{1}{268}\right)+122\arctan\left(\frac{1}{343}\right)+\\
&+115\arctan\left(\frac{1}{557}\right)-32\arctan\left(\frac{1}{1068}\right)\\
&+83\arctan\left(\frac{1}{3458}\right)+44\arctan\left(\frac{1}{27493}\right) \qquad \text{Ärndt}
\end{aligned}$$

La meilleure solution est, en théorie, la dernière, puisque c'est celle qui dispose des facteurs les plus petits. Mais nous verrons en pratique qu'elles sont pratiquement équivalentes au niveau temps de calcul.

On trouvera également dans [Wei] de nombreuses autres formules qui pourraient donner une complexité plus faible.

5 Résultats

Les méthodes ci-dessus ayant une bonne complexité, elles ont (souvent) surpassé les techniques présentes dans les anciennes version de mpfr. Ceci est assez visible dans le cas de l'exponentielle et de pi. Pour sinus et cosinus, aucune comparaison n'était possible. Les tests ont tous été réalisés sur un Pentium III 500 Mhz sous Linux, les temps sont donnés en millisecondes. Les résultats sont similaires sur un Alpha OSF, mais meilleurs en temps de calcul.

5.1 L'exponentielle

Les résultats sont ici tout à fait probants, puisque l'utilisation de l'algorithme fournit ici des résultats bien meilleurs. On vérifiera bien ici la complexité théorique indiquée. `mpfr_exp2` correspond à l'algorithme précédemment implanté, dont le temps de calcul est en $O(M(n))n^{1/3}$.

n(bits)	`mpfr_exp2`	`mpfr_exp3`	*ratio*
50	0.03	0.06	2.00
100	0.06	0.1	1.66
1000	0.8	1.0	1.25
2000	2.8	3.3	1.17
5000	15.2	17.2	1.13
10000	50	50	1.00
20000	190	180	0.97
50000	1050	710	0.67
100000	3700	2210	0.59
200000	13020	6500	0.50
500000	67960	24680	0.36

TAB. 3 – *Calcul de l'exponentielle*

n	temps t(ms)	$\frac{t \cdot 10^5}{\mathrm{M}(n)\log(n)}$
2000	3.3	1.6
5000	17.2	2
10000	50	1.98
20000	180	2.4
50000	710	2.3
100000	2210	2.48
200000	6500	2.52
500000	24680	2.36

TAB. 4 – *Complexité temporelle théorique de l'exponentielle*

5.2 $\ln 2$

Les résultats sont ici plus mitigés. En effet, il est de toute façon difficile d'avoir un bon algorithme pour calculer $\ln 2$, et l'utilisation des séries hypergéométriques montre ici ses limites. Néanmoins, à partir de 100000 chiffres, l'algorithme est plus performant.

5.3 π

Là encore, les résultats sont plutôt bons, puisqu'on arrive à être dix fois plus rapide pour 500000 chiffres, ce qui montre bien que la méthode est meilleure.

n	Standard	Algorithme générique		
		Formule 1	Formule 3	Formule 4
5000	20	110	70	50
10000	80	340	230	150
20000	290	1110	730	440
50000	1790	3450	2300	1400
100000	7900	10730	7330	4450
200000	33140	33240	22660	13720
500000	208790	100720	69730	43070

TAB. 5 – *Calcul de* $\ln 2$

n	Standard	Algorithme générique		
		Machin	Gauss	Arndt
5000	20	33	30	31
10000	70	100	90	90
20000	280	310	270	280
50000	1720	990	900	900
100000	7980	3050	2780	2750
200000	38230	9720	8860	8700
500000	279340	30400	28210	28220

TAB. 6 – *Calcul de* π

Remarque : Certains des résultats obtenus peuvent être améliorés en modifiant légèrement l'algorithme pour qu'il s'arrête dès que la précision nécessaire a été atteinte, et non lorsqu'il arrive sur une puissance de 2. Ceci induit en effet un phénomène en escalier qui n'a pas lieu d'être. Cette optimisation a déjà été menée pour l'exponentielle, mais ne l'est pas encore sur l'algorithme générique.

6 Conclusion

Grâce à l'utilisation des fonctions hypergéométriques et à la méthode *binary splitting*, la vitesse de calcul de certaines fonctions de mpfr a pu être augmentée, sans pour autant sacrifier en précision. On pourra ensuite réutiliser les mêmes algorithmes pour calculer d'autres fonctions, voire d'autres constantes.

Références

[BB87] Jonathan M. Borwein and Peter B. Borwein. *Pi and The AGM.* Wiley-Interscience, 1987.

[Bre76] Richard P. Brent. The complexity of multiple-precision arithmetic. *The Complexity of Computational Problem Solving*, 1976.

[Gra00] Torbjörn Granlund. *The GNU Multiple Precision Arithmetic Library*, 3.0.1 edition, April 2000. URL: `http://www.swox.com/gmp/`.

[HP97] Bruno Haible and Thomas Papanikolaou. Fast multiprecision evaluation of series of rational numbers. 1997.

[Kar91] Ekatharine A. Karatsuba. Fast evaluation of transcendal functions. *Problemy Peredachi Informatsii*, 1991.

[Kar97] Ekatharine A. Karatsuba. Fast evaluation of hypergeometric functions by FEE. *Computational Methods and Function Theory*, 1997.

[Pol99] PolKA. *The Multiple Precision Floating-Point Library*, 1.0 edition, June 1999. Disponible à l'URL: `http://www.loria.fr/projets/mpfr`.

[Wei] Eric Weisstein. Database of Machin-like formulas. In *Eric Weisstein's World of Mathematics.* URL : `http://mathworld.wolfram.com/notebooks/MachinFormulas.m`.

Table des matières

Liste des tableaux

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803